Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
«Южно-Уральский государственный университет»
(Национальный исследовательский университет)
Факультет вычислительной математики и информатики
Кафедра экономико-математических методов и статистики

РАБОТА ПРОВЕРЕНА
Рецензент,

_____

« » _____ 2015 г.

ДОПУСТИТЬ К ЗАЩИТЕ
Заведующий кафедрой, д. ф.-м. н., профессор
_____ А.В. Панюков

« » _____ 2015 г.

Параллельная реализация метода эллипсоидов для задач оптимизации большой размерности

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ ЮУрГУ-010400.62.2015.001.001 ВКР

Консультант,
_____

« » _____ 2015 г.

Руководитель проекта,
_____ В.А. Голодов

« » _____ 2015 г.

Автор проекта
студент группы ВМИ-413
_____ В.А. Безбородов
« » _____ 2015 г.

Нормоконтролер, к. ф.-м. н., доцент
_____ Т.А. Макаровских
« » _____ 2015 г.

Челябинск, 2015

Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
«Южно-Уральский государственный университет»
(Национальный исследовательский университет)
Факультет вычислительной математики и информатики
Кафедра экономико-математических методов и статистики

УТВЕРЖДАЮ
Заведующий кафедрой, д. ф.-м. н., профессор
_____ А.В. Панюков
« » _____2015 г.

З А Д А Н И Е

на выпускную квалификационную работу студента

Безбородова Вячеслава Александровича

Группа ВМИ-413

1. Тема работы

   Параллельная реализация метода эллипсоидов для задач оптимизации большой размерности

   утверждена приказом по университету от « » _____2015 г. №_____

2. Срок сдачи студентом законченной работы « » _____2015 г.
3. Исходные данные к работе

   3.1. Данные из учебной литературы;

   3.2. Самостоятельно сконструированные тестовые данные.

4. Перечень вопросов, подлежащих разработке

   4.1. Изучение общей схемы работы метода эллипсоидов;

   4.2. Изучение приемов параллельной обработки данных;

   4.3. Разработка класса (типа данных) для реализации параллельно выполняемых операций над матрицами с применением библиотеки GMP;

4.4. Разработка параллельной реализации метода эллипсоидов для задачи линейного программирования;

4.5. Оценка сложности полученной реализации;

4.6. Сравнение с известными методами решения;

4.7. Тестирование;

4.8. Проверка на модельных данных.

5. Иллюстративный материал

5.1. Энергетическо-трудовой цикл 1л.

5.2. Объект, предмет и цель дипломной работы 1л.

5.3. Задачи дипломной работы 1л.

5.4. Новая концепция управления экономическими системами 1л.

5.5. Концептуальная схема модели 1л.

5.6. Общий вид модели в среде VisSim 1л.

5.7. Элементы модели энергетическо-трудового цикла 1л.

5.8. Система уравнений модели животноводства 2л.

5.9. Принцип управления 1л.

5.10. Результаты эксперимента 10л.

5.11. Заключение 1л.

5.12. Благодарность за внимание 1л.

6. Календарный план

Наименование этапов дипломной работы	Срок выполнения этапов работы	Отметка о выполнении
1. Сбор материалов и литературы по теме дипломной работы	02.02.2015 г.	
2. Исследование способов построения математической модели задачи		
3. Разработка математической модели и алгоритма		
4. Реализация разработанных алгоритмов		
5. Проведение вычислительного эксперимента		
6. Подготовка пояснительной записки дипломной работы		
Написание главы 1		
Написание главы 2		
Написание главы 3		
7. Оформление пояснительной записки		
8. Получение отзыва руководителя		
9. Проверка работы руководителем, исправление замечаний		
10. Подготовка графического материала и доклада		
11. Нормоконтроль		
12. Рецензирование, представление зав. кафедрой	10.06.2015 г.	

7. Дата выдачи задания « » 2015 г.

Заведующий кафедрой_____/А.В. Панюков /

Руководитель работы_____/В.А. Голодов /

Студент_____/В.А. Безбородов /

Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
«Южно-Уральский государственный университет»
(Национальный исследовательский университет)
Факультет вычислительной математики и информатики
Кафедра экономико-математических методов и статистики

АННОТАЦИЯ

Безбородов, В.А. Параллельная реализация метода эллипсоидов для задач оптимизации большой размерности / В.А. Безбородов . – Челябинск: ЮУрГУ, Факультет вычислительной математики и информатики, 2015 . – 38 с., 1 ил., 3 табл., библиогр. список – 29 названий.

В дипломной работе произведен анализ алгоритма метода эллипсоидов на предмет вычислительной сложности выполняемых операций. На основе результатов анализа разработана параллельная реализация метода эллипсоидов, адаптированная для решения задач оптимизации большой размерности на многопроцессорных и/или многоядерных вычислительных системах с общей разделяемой памятью.

Приведены результаты вычислительных экспериментов, продемонстрирован пример решения задачи оптимизации большой размерности с применением разработанной программной реализации.

ОГЛАВЛЕНИЕ

Введение

Задачи оптимизации получили чрезвычайно широкое распространение в технике, экономике, управлении. Типичными областями применения теории оптимизации являются прогнозирование, планирование промышленного производства, управление материальными ресурсами, а также контроль качества выпускаемой продукции [16].

Успешность хозяйственной деятельности зависит от того, как распределяются имеющиеся ограниченные ресурсы. В связи с тем, что такая задача оптимального распределения довольно часто возникает на практике в различных сферах жизнедеятельности, актуальным становится поиск способов ускорения ее решения.

Задачи оптимизации большой размерности характеризуются высокой трудоемкостью. Использование доступного ресурса аппаратного параллелизма современных вычислительных систем рассматривается как возможность ускорения поиска их решения. Применение библиотек, реализующих поддержку арифметики произвольной точности, диктуется необходимостью достижения высокой точности при решении практических задач.

Разрабатывали, а впоследствии развивали метод эллипсоидов такие ученые, как Шор Н.З. [27], Юдин Д.Б., Немировский А.С. [29], Хачиян Л.Г. [22], Гершович В.И. [14], Стецюк П.И. [21] и др.

В работе исследован алгоритм метода эллипсоидов. Разработана его программная реализация, ориентированная на многопроцессорные и/или многоядерные вычислительные системы с общей разделяемой памятью. Показано приложение программной реализации к решению задачи оптимизации большой размерности.

Целями работы являются:

1) разработка параллельной реализации метода эллипсоидов, поддерживающей арифметику произвольной точности;
2) использование полученной реализации метода эллипсоидов для решения задачи оптимизации большой размерности.

В соответствии с поставленными целями в работе решаются следующие **задачи**:

- Исследование операций классического алгоритма метода эллипсоидов на вычислительную сложность;
- Разработка программной реализации алгоритма с распараллеливанием наиболее длительных по времени операций;
- Обеспечение поддержки арифметики расширенной и произвольной точности;
- Проверка и тестирование разработанного программного обеспечения.

Объектом исследования данной работы является метод эллипсоидов, **предметом** – параллельная реализация метода, поддерживающая арифметику произвольной точности.

Работа состоит из введения, 2 глав, заключения и списка литературы. Объем работы составляет 38 страниц. Список литературы содержит 29 наименований.

В первой главе рассматривается алгоритм метода эллипсоидов, производится его анализ на предмет вычислительной сложности с целью поиска наиболее ресурсоемких операций, нуждающихся в ускорении путем распараллеливания.

Во второй главе приведены данные, необходимые для...

В третьей главе приводится краткое описание...

В четвертой главе построена...

В пятой главе приводятся результаты вычислительных экспериментов...

В заключении перечислены основные результаты работы.

1 Метод эллипсоидов

В настоящее время большое внимание уделяется созданию автоматизированных систем планирования, проектирования и управления в различных областях промышленности. На первый план выдвигаются вопросы качества принимаемых решений, в связи с чем возрастает роль методов и алгоритмов решения оптимизационных задач в математическом обеспечении автоматизированных систем различного уровня и назначения.

Имеется несколько основных источников, порождающих задачи оптимизации: задачи математического программирования, задачи нелинейного программирования, задачи оптимального управления, задачи дискретного программирования или задачи смешанного дискретно-непрерывного типа [28].

Сфера применения методов оптимизации огромна. Создание эффективных методов оптимизации является ключом к решению многих вычислительных проблем математического программирования, особенно для задач большой размерности.

Для минимизации гладких функций широко применяются различные модификации градиентных процессов, поскольку направление антиградиента в данной точке локально является направлением наискорейшего спуска. Регулировка шага в большинстве алгоритмов этого типа основана на том, чтобы обеспечить монотонное и в достаточной степени «существенное» уменьшение значения функции на каждом шаге.

Простейший обобщенный градиентный метод состоит в движении на каждом шаге в направлении, обратном направлению обобщенного градиента. Этот метод под названием обобщенного градиентного спуска (ОГС) предложен Н.З. Шором в 1961 г. в связи с необходимостью разработки эффективного алгоритма решения транспортных задач большой размерности для задач текущего планирования, решаемых в Институте кибернетики АН УССР совместно с Госпланом УССР. Впервые метод обобщенного градиентного спуска для минимизации кусочно-линейных выпуклых функций использовался при решении транспортных и транспортно-производственных задач [23]. Затем метод ОГС был распространен на класс произвольных выпуклых функ-

ций [24] и на задачи выпуклого программирования в гильбертовом пространстве [19]. Широкое распространение получили стохастические аналоги ОГС [17]. Алгоритмы решения задач математического программирования, построенные на основе ОГС, отличаются простотой и, что особенно важно для задач большой размерности, экономным использованием оперативной памяти ЭВМ.

К ограничениям метода ОГС относятся его довольно медленная сходимость, сложность контроля точности решения и то, что он применим только к классу выпуклых функций.

В 1969–1970 гг. Н.З. Шором были предложены ускоренные варианты обобщенных градиентных методов, основанные на использовании операции растяжения пространства в направлении градиента и разнсти двух последовательных градиентов [25]. Идея этих методов существенно отлична от той, которая используется для ускорения сходимости в случае гладких функций – идеи квадратичной аппроксимации функции в окрестности минимума, в той или иной мере определяющей формализм как методов сопряженных градиентов, так и квазиньютоновских методов [6]. В то же время предельные варианты методов с растяжением пространства при определенных условиях регулярности и гладкости обладают свойством квадратичной скорости сходимости. Таким образом, предложенные алгоритмы обладают высокой эффективностью и применительно к гладким задачам минимизации. В дальнейшем алгоритмы с растяжением пространства были обобщены на задачи нахождения локальных минимумов невыпуклых негладких функций [26].

В США и Западной Европе градиентными методами минимизации негладких функций всерьез начали заниматься примерно с 1973 г. сначала в связи с приложениями в области дискретного программирования [4], а затем в целом для решения задач большой размерности [5]. Результаты работ в этом направлении на Западе достаточно полно представлены в сборнике [8]. Особенно интенсивно развивается направление так называемой ε-субградиентной оптимизации, по идее близкое, с одной стороны, к алгоритмам В.Ф. Демьянова решения минимаксных задач, а с другой, особенно в формальном отноше-

нии, – к алгоритмам метода сопряженных градиентов (или «давидоновского» типа).

И наконец, в последнее время обнаружились очень интересные связи между алгоритмами последовательных отсечений и алгоритмами с растяжением пространства [27, 29].

Таким образом, область обобщенных градиентных методов оптимизации не представляет нечто окончательно сформировавшееся и застывшее, а, наоборот, быстро развивается.

1.1 Алгоритм метода эллипсоидов

Рассмотрим алгоритм решения задачи выпуклого программирования, гарантирующий уменьшение объема области, в которой локализуется оптимум, со скоростью геометрической прогрессии, причем знаменатель этой прогрессии зависит только от размерности задачи. Этот алгоритм относится к классу алгоритмов обобщенного градиентного спуска с растяжением пространства в направлении градиента (ОГСРП) [27].

Пусть имеется задача выпуклого программирования:

$$\min f_0(x) \tag{1.1}$$

при ограничениях

$$f_i(x) \leq 0, \quad i = 1, \ldots, m, \quad x \in E_n, \tag{1.2}$$

где E_n – евклидово пространство размерности n, $f_\nu(x)$, $\nu = \overline{0, m}$ – выпуклые функции, определенные на E_n; $g_\nu(x)$ – субградиенты соответствующих функций. Пусть имеется априорная информация о том, что существует оптимальная точка $x^* \in E_n$ (не обязательно единственная), которая находится в шаре радиуса R с центром в точке x_0 (формально к системе ограничений 1.2 можно добавить ограничение $||x - x_0|| \leq R$).

Рассмотрим следующий итеративный алгоритм (при $n > 1$).

Алгоритм 1 Метод эллипсоидов

Шаг 0. Инициализация.
Положить $x_k = x_0$; $B_k = E$, где E – единичная матрица размерности $n \times n$; $h_k = \frac{R}{n+1}$ – коэффициент, отвечающий за уменьшение объема шара. Перейти к шагу 1.
Шаг 1. Вычислить

$$g(x_k) = \begin{cases} g_0(x_k), & \text{если } \max\limits_{1 \le i \le m} f_i(x_k) \le 0, \\ g_{i^*}(x_k), & \text{если } \max\limits_{1 \le i \le m} f_i(x_k) = f_{i^*}(x_k) > 0. \end{cases}$$

Если $g(x_k) = 0$, то завершить алгоритм; x_k – оптимальная точка. Иначе перейти к шагу 2.
Шаг 2. Вычислить $\xi_k = \dfrac{B_k^T g(x_k)}{||B_k^T g(x_k)||}$. Перейти к шагу 3.
Шаг 3. Вычислить $x_{k+1} = x_k - h_k \cdot B_k \cdot \xi_k$. Перейти к шагу 4.
Шаг 4. Вычислить $B_{k+1} = B_k \cdot R_\beta(\xi_k)$, где $R_\beta(\xi_k)$ – оператор растяжения пространства в направлении ξ_k с коэффициентом β (см. определение 1.1), $\beta = \sqrt{\frac{n-1}{n+1}}$. Перейти к шагу 5.
Шаг 5. Вычислить $h_{k+1} = h_k \cdot r$, где $r = \frac{n}{\sqrt{n^2-1}}$. Перейти к шагу 1.

Определение 1.1. Оператором растяжения пространства E_n в направлении ξ с коэффициентом β называется оператор $R_\beta(\xi)$, действующий на вектор x следующим образом [28]:

$$R_\beta(\xi)x = \left(E - (\beta - 1)\xi_k \xi_k^T\right) x.$$

Рассмотрим вопрос оценки скорости сходимости метода эллипсоидов. Покажем, что данный вариант алгоритма ОГСРП сходится по функционалу со скоростью геометрической прогрессии, причем знаменатель этой прогрессии зависит только от размерности задачи.

Лемма 1.1. Последовательность $\{x_k\}_{k=0}^{\infty}$, генерируемая алгоритмом 1, удовлетворяет неравенству

$$||A_k(x_k - x^*)|| \le h_k \cdot (n+1), \quad A_k = B_k^{-1}, \quad k = 0, 1, 2, \ldots \tag{1.3}$$

Доказательство леммы для краткости изложения опущено и может быть найдено в [27].

Множество точек x, удовлетворяющих неравенству

$$||A_k(x_k - x)|| \leq (n+1)h_k = R \cdot \left(\frac{n}{\sqrt{n^2-1}}\right)^k,$$

представляет собой эллипсоид Φ_k, объем которого $v(\Phi_k)$ равен

$$\frac{v_0 R^n \left(\frac{n}{\sqrt{n^2-1}}\right)^{nk}}{\det A_k},$$

где v_0 – объем единичного n-мерного шара. Получаем

$$\frac{v(\Phi_{k+1})}{(\Phi_k)} = \frac{\left(\frac{n}{\sqrt{n^2-1}}\right)^n \cdot \det A_k}{\det A_{k+1}} = \frac{\left(\frac{n}{\sqrt{n^2-1}}\right)^n \cdot \det A_k}{\det R_\alpha(\xi_k) \cdot \det A_k} = \frac{1}{\alpha}\left(\frac{n}{\sqrt{n^2-1}}\right)^n =$$

$$= \sqrt{\frac{n-1}{n+1}}\left(\frac{n}{\sqrt{n^2-1}}\right)^n = q_n < 1.$$

Таким образом, объем эллипсоида, в котором локализуется оптимальная точка x^* в соответствии с неравенством (1.3), убывает со скоростью геометрической прогрессии со знаменателем q_n.

1.2 Вычислительная сложность операций метода эллипсоидов

В индустрии разработки программного обеспечения известен феномен, который состоит в том, что на 20% методов программы приходится 80% времени ее выполнения [1]. Такое эмпирическое правило хорошо согласуется с более общим принципом, известным как принцип Парето либо «правило 80/20».

Утверждение 1 (Принцип Парето). 80% результата можно получить, приложив 20% усилий.

Относящийся не только к программированию, этот принцип очень точно характеризует оптимизацию программ [18]. В работе [7] Дональд Кнут указал,

что менее 4% кода обычно соответствуют более чем 50% времени выполнения программы. Опираясь на принцип Парето, можно сформулировать последовательность действий, приводящих к ускорению работы имеющихся программ: необходимо найти в коде «горячие точки» и сосредоточиться на оптимизации наиболее трудоемких процессов.

Проанализируем подробнее вычислительную сложность операций алгоритма метода эллипсоидов. Для этого введем некоторые обозначения из области асимптотического анализа [15].

Определение 1.2. Функция $f(n)$ *ограничена сверху* функцией $g(n)$ асимптотически с точностью до постоянного множителя, т.е.

$$f(n) = O\left(g(n)\right),$$

если существуют целые N и K, такие, что $|f(n)| \leq Kg(n)$ при всех $n \geq N$.

Определение 1.3. Функция $f(n)$ *ограничена снизу* функцией $g(n)$ асимптотически с точностью до постоянного множителя, т.е.

$$f(n) = \Omega\left(g(n)\right),$$

если существуют целые N и K, такие, что $f(n) \geq Kg(n)$ при всех $n \geq N$.

Определение 1.4. Функция $f(n)$ *ограничена снизу и сверху* функцией $g(n)$ асимптотически с точностью до постоянного множителя, т.е.

$$f(n) = \Theta\left(g(n)\right),$$

если одновременно выполнены условия определений 1.2 и 1.3.

На **шаге 1** алгоритма 1 изменение текущего субградиента $g(x_k)$ происходит на основании анализа значений функций ограничений (1.2) в текущей точке x_k, т.е. анализируется последовательность значений

$$\max_{1 \leq i \leq m} f_i(x_k).$$

Вычислительная сложность поиска максимального из m чисел зависит от выбора алгоритма.

При использовании линейного последовательного поиска цикл выполнит m итераций. Трудоемкость каждой итерации не зависит от количества элементов, поэтому имеет сложность $T^{iter} = O(1)$. В связи с этим, верхняя оценка всего алгоритма поиска $T_m^{min} = O(m) \cdot O(1) = O(m \cdot 1) = O(m)$. Аналогично вычисляется нижняя оценка сложности, а в силу того, что она совпадает с верхней, можно утверждать $T_m^{min} = \Theta(m)$.

При использовании алгоритма бинарного поиска на каждом шаге количество рассматриваемых элементов сокращается в 2 раза. Количество элементов, среди которых может находиться искомый, на k-ом шаге определяется формулой $\frac{m}{2^k}$. В худшем случае поиск будет продолжаться, пока в массиве не останется один элемент, т.е. алгоритм имеет логарифмическую сложность: $T_m^{binSearch} = O(\log(m))$. Резюмируем все вышесказанное относительно алгоритмов поиска в виде таблицы 1.

Таблица 1 — Оценка сложности некоторых алгоритмов поиска

Алгоритм	**Структура данных**	**Временная сложность**		**Сложность по памяти**
		В среднем	**В худшем**	**В худшем**
Линейный поиск	Массив из n элементов	$O(n)$	$O(n)$	$O(1)$
Бинарный поиск	Отсортированный массив из n элементов	$O(\log(n))$	$O(\log(n))$	$O(1)$

Однако при использовании некоторых алгоритмов (например, алгоритма бинарного поиска) потребуются дополнительные процедуры для упорядочивания входной последовательности значений. В таблице 2 представлены асимптотические оценки наиболее известных алгоритмов сортировки массива из n элементов[1].

[1]Более подробный анализ приведен в [13].

Таблица 2 — Оценка сложности некоторых алгоритмов сортировки[1]

Алгоритм	**Временная сложность**			**Сложность по памяти**
	В лучшем	**В среднем**	**В худшем**	**В худшем**
Быстрая сортировка	$O(n\log(n))$	$O(n\log(n))$	$O(n^2)$	$O(n)$
Сортировка слиянием	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$	$O(n)$
Пирамидальная сортировка	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$	$O(1)$
Пузырьковая сортировка	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Сортировка вставками	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Сортировка выборомм	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Блочная сортировка	$O(n+k)$	$O(n+k)$	$O(n^2)$	$O(nk)$
Поразрядная сортировка	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$

На **шаге 2** алгоритма производится вычисление нормированного обобщенного градиента

$$\xi_k = \frac{B_k^T g(x_k)}{||B_k^T g(x_k)||},$$

что подразумевает выполнение трудоемких матричных операций, таких как транспонирование, умножение на вектор и на число. Асимптотическая сложность таких операций для матрицы размерности $n \times n$ может быть оценена как $O(n^2)$.

На **шаге 3** при обновлении значения текущей точки выполняется умножение матрицы B_k на вектор ξ_k:

$$x_{k+1} = x_k - h_k \cdot B_k \cdot \xi_k.$$

Аналогично предыдущей оценке, цена такой операции составит $O(n^2)$.

Шаг 4 наиболее сложен из-за необходимости вычисления оператора растяжения пространства

$$B_{k+1} = B_k \cdot R_\beta(\xi_k).$$

Если представить оператор в матричной форме (определение 1.1), то можно видеть, что в процессе его вычисления используются все вышеперечисленные операции, а также операция сложения матриц, сложность которой составляет $O(n^2)$.

На **шаге 5** осуществляется пересчет коэффициента h_k, отвечающего за уменьшение объема шара

$$h_{k+1} = h_k \cdot r.$$

Эта операция может быть выполнена за константное время, т.е. асимптотически ее сложность составит $O(1)$.

Из проведенного анализа вычислительной сложности операций, входящих в алгоритм метода эллипсоидов, можно сделать несколько выводов. Во-первых, учитывая специфику рассматриваемого класса задач (задачи оптимизации большой размерности), наиболее трудоемкие операции будут выполняться значительно дольше менее трудоемких, что приведет к сильно неравномерной загрузке вычислительной системы. Во-вторых, схема чередования сложных/простых в вычислительном смысле операций, а также выбор целевой платформы (многопроцессорные и/или многоядерные системы с общей разделяемой памятью) наталкивают на возможность использования Fork-Join Model (FJM) модели распараллеливания задач для ускорения работы алгоритма метода эллипсоидов.

1.3 Парадигма Fork-Join

В параллельном программировании, Fork-Join model (модель ветвление-объединение, FJM) – это способ запуска и выполнения параллельных участков кода, при котором выполнение ветвей завершается в специально обозначенном месте для того, чтобы в следующей точке продолжить последовательное выполнение. Параллельные участки могут разветвляться рекурсивно до тех пор, пока не будет достигнута заданная степень гранулярности задачи. Модель была впервые сформулирована в 1963 г., и может рассматриваться как один из параллельных паттернов проектирования [9].

Различные реализации FJM обычно управляют *задачами*, *волокнами* или *легковесными нитями*, а не *процессами* уровня операционной системы, и используют *пул потоков* для их выполнения. Специальные ключевые слова позволяют программисту определять точки *возможного параллелизма*; проблема создания реальных потоков и управления ими ложится на реализацию.

В [2] приведена иллюстрация парадигмы Fork-Join. Представим ее на рисунке 1. Здесь три участка программы потенциально разрешают параллельное исполнение различных блоков. Последовательное выполнение показано сверху, в то время как его Fork-Join эквивалент снизу.

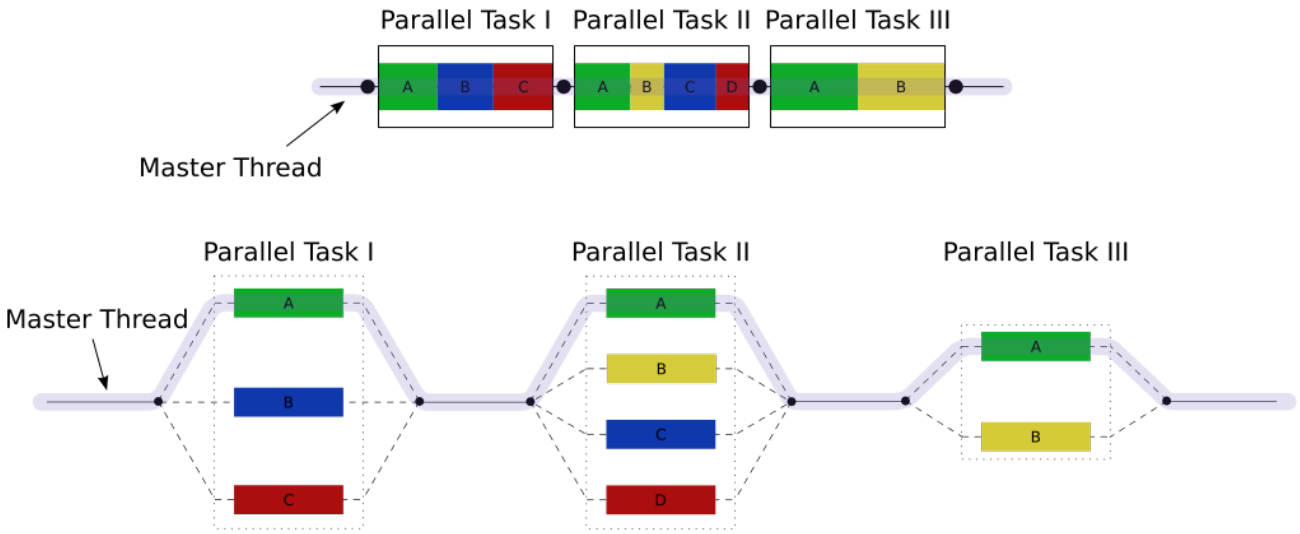

Рисунок 1 — Иллюстрация парадигмы Fork-Join

Легковесные нити, используемые в Fork-Join программировании, обычно имеют свой собственный планировщик, который управляет ими, применяя схему пула потоков. Такой планировщик может быть гораздо проще полнофункционального планировщика задач, применяемого операционной системой. Планировщики потоков общего пользования обязаны приостанавливать-/запускать потоки в обозначенных программистом местах, в то время как в парадигме Fork-Join потоки блокируются только в местах непосредственного слияния.

Модель Fork-Join – основная модель параллельного исполнения в технологии OpenMP. Также эта модель поддерживается в Java concurrency framework, в Task Parallel Library для .NET и в Intel Threading Building Blocks. Языки программирования Cilk и Cilk Plus имеют встроенную под-

держку FJM в форме ключевых слов *spawn–sync* и *cilk_spawn–cilk_sync* соответственно.

Выводы по главе один

Как было показано, широта и разнообразие применения методов оптимизации обуславливают необходимость создания эффективных методов оптимизации для решения различных вычислительных проблем математического программирования.

Для минимизации гладких функций широко применяются различные методы. В этой главе были подробно рассмотрены методы обобщенного градиентного спуска (ОГС) и обобщенного градиентного спуска с растяжением пространства в направлении градиента (ОГСРП), предложенные Н.З. Шором.

Было доказано, что метод эллипсоидов гарантированно локализует оптимум со скоростью геометрической прогрессии при условии существования оптимальной точки.

Проведенный анализ метода эллипсоидов показал, что некоторые операции метода вычислительно сложны. Была выдвинута гипотеза о том, что на современных многопроцессорных и/или многоядерных системах с общей разделяемой памятью можно добиться ускорения работы всего метода, если распараллелить наиболее ресурсоемкие операции. В главе дается подробное объяснение почему для этой цели лучше всего подходит модель Fork-Join параллельного запуска участков кода.

Во второй главе описывается способ параллельной реализации метода эллипсоидов, аргументированно доказывается необходимость обеспечения поддержки арифметики расширенной точности, а также описывается тестовое окружение разрабатываемого гласса.

2 Параллельная реализация метода эллипсоидов

В первой главе операции метода эллипсоидов были подробно проанализированы на предмет вычислительной сложности. На основании полученных данных можно сформулировать гипотезу о том, насколько удастся ускорить выполнение метода в целом для решения задач оптимизации большой размерности, если к наиболее ресурсоемким операциям применить алгоритм распараллеливания по данным.

Гипотеза 1 (О соотношении времени). Пусть $f(t)$ – это время работы алгоритма метода эллипсоидов для задачи оптимизации размерности $N \times M$, выполняемого *в однопоточном режиме*. Тогда для параллельной реализации метода эллипсоидов, выполняемой *в многопоточном режиме*, для достаточно больших N и M справедливо равенство

$$F(t) = kf(t),$$

где $F(t)$ – общее время работы параллельной реализации метода, а k – коэффициент ускорения ($k > 1$).

Приближенная оценка для коэффициента ускорения k может быть получена в ходе выполнения вычислительных экспериментов.

В процессе решения любой задачи оптимизации возникает необходимость оперирования над матрицами, которые составляются исходя из условий конкретной задачи. Поскольку в данной работе изначально заложена ориентация на решение задач оптимизации большой размерности, то и матрицы, возникающие из анализа условий этих задач, будут иметь большую размерность. Любое численное решение должно достигать определенной точности результата. А при оперировании матрицами большой размерности обеспечение требуемой точности становится еще более актуальным.

Один из способов обеспечения поддержки арифметики расширенной точности заключается в использовании готовых специализированных математических библиотек, таких, например, как GMP.

2.1 Необходимость использования библиотеки GMP

Библиотека GMP [3] – *бесплатная (свободная)* библиотека для арифметики произвольной точности, выполняемой над знаковыми целыми, рациональными числами и числами с плавающей запятой. При этом практически не существует предела для точности вычислений, если не считать объем доступной памяти ЭВМ, на которой производятся вычисления. GMP имеет богатый набор функций, которые имеют стандартизированный интерфейс.

В основном GMP применяется в криптографических, научно-исследовательских приложениях, приложениях, отвечающих за безопасность в сети Интернет, различных системах вычислительной алгебры и т.д.

Основными целевыми платформами GMP являются Unix-подобные системы, такие как GNU/Linux, Solaris, HP-UX, Mac OS X/Darwin, BSD, AIX и проч. Также поддерживается работа на Windows в 32 и 64-битном режимах.

Библиотека GMP тщательно спроектирована для того, чтобы вычисления производились настолько быстро, насколько это возможно одновременно и для больших, и для малых операндов. Такая скорость возможна благодаря использованию машинных слов в качестве базового арифметического типа и быстрых алгоритмов, включающих высокооптимизированный ассемблерный код для большинства внутренних циклов для целого набора наиболее популярных современных центральных процессоров.

Первая версия GMP вышла в 1991 году. С тех пор библиотека непрерывно улучшается и поддерживается, обеспечивая выход новых версий примерно раз в год.

Начиная с версии 6, GMP распространяется одновременно под двумя лицензиями: GNU LGPL v3 и GNU GPL v2. Такое лицензирование позволяет использовать библиотеку бесплатно, изменять ее и публиковать результат.

Приведем численный пример, наглядно доказывающий необходимость использования библиотеки GMP для поддержки арифметики расширенной и произвольной точности. Рассмотрим следующий код, представленный в листинге 1.

```
1  MatrixRandomFiller filler;
2  MatrixMultiplier multiplier;
```

```
3   MatrixPrinter printer( &std::cout );
4   printer.setPrecision( 20 );
5
6   std::size_t nRows = 2;
7   std::size_t nColumns = 2;
8
9   Matrix< double > matrix_a( nRows, nColumns );
10  Matrix< double > matrix_b( nRows, nColumns );
11
12  filler.fill( matrix_a );
13  filler.fill( matrix_b );
14
15  printer.print( matrix_a );
16  printer.print( matrix_b );
17
18  Matrix< double > matrix_c( nRows, nColumns );
19  multiplier.multiply( matrix_c, matrix_a, matrix_b );
20
21  Matrix< double > matrix_d( nRows, nColumns );
22  matrix_d = matrix_a * matrix_b;
23
24  printer.print( matrix_c );
25  printer.print( matrix_d );
```

Листинг 1 — Исходный код примера

В листинге используются объекты следующих классов:

1) `Matrix< Type >` представляет в программе матрицу, элементы которой имеют тип `Type`;
2) `MatrixRandomFiller` предназначен для заполнения матрицы случайными значениями, равномерно распределенными на интервале $[0, 1)$ (в процессе заполнения используются стандартный генератор псевдослучайных чисел `std::default_random_engine` и равномерное распределение `std::uniform_real_distribution` $P(i|a, b) = \frac{1}{b-a}$ с параметрами $a = 0$, $b = 1$);
3) `MatrixMultiplier` производит перемножение матриц;
4) `MatrixPrinter` отвечает за печать матриц.

В строках 1–4 создаются и настраиваются следующие вспомогательные объекты: `filler`, `multiplier` и `printer`. `Printer` настраивается таким

образом, чтобы осуществлять вывод в стандартный поток вывода `std::cout` и печатать матрицы с точностью до 20 знаков после запятой. В строках 6–7 задается размерность матриц, участвующих в данном примере. Затем происходит создание (строки 9–10), заполнение случайными значениями (строки 12–13) и печать (строки 15–16) двух операндов. В строке 19 осуществляется перемножение матриц `matrix_a` и `matrix_b` стандартным базовым алгоритмом (строка на столбец), результат которого сохраняется в объекте `matrix_c`. В строке 22 также осуществляется перемножение операндов. Отличие от строки 19 состоит в том, что в данном случае для перемножения используется функция-член класса `Matrix`. Результат второго перемножения сохраняется в переменной `matrix_d`. После этого осуществляется печать полученных результатов (строки 24–25).

Поскольку матрицы заполняются случайно, вывод программы зависит от конкретного запуска. В частности, возможен и такой вариант вывода.

$$matrix\_a = \begin{pmatrix} 0.54846871850742517918 & 0.84412510404456542190 \\ 0.95768018538410970564 & 0.80660421324152631328 \end{pmatrix},$$

$$matrix\_b = \begin{pmatrix} 0.96126112219013459814 & 0.79203852615307757112 \\ 0.79959221481843290036 & 0.90154689901948148467 \end{pmatrix},$$

$$matrix\_c = \begin{pmatrix} 1.20217751736546674124 & 1.19542672538356331557 \\ 1.565535179049257\underline{09736} & 1.48571112974158303643 \end{pmatrix},$$

$$matrix\_d = \begin{pmatrix} 1.20217751736546674124 & 1.19542672538356331557 \\ 1.565535179049256\underline{87532} & 1.48571112974158303643 \end{pmatrix}.$$

Из приведенных значений элементов матриц хорошо видно, что для одних и тех же операндов результат перемножения, полученный вне и внутри класса `Matrix`, различен, начиная с 15-го знака после запятой. При этом популярный математический сервис WolframAlpha [12] для данных операндов выдает результат

$$\begin{pmatrix} 1.2021775173654667\underline{500} & 1.1954267253835633\underline{424} \\ 1.5655351790492\underline{57001}1 & 1.4857111297415\underline{829851} \end{pmatrix},$$

в котором наблюдаются отличия в знаках во всех элементах матрицы. Попытка провести аналогичные вычисления на языке программирования R приводит к результату, показанному в листинге 2.3.

```
> options(digits=20)
> lhs = matrix( c( 0.54846871850742517918, 0.84412510404456542190,
                   0.95768018538410970564, 0.80660421324152631328 ),
                nrow=2, ncol=2, byrow=TRUE )
> rhs = matrix( c( 0.96126112219013459814, 0.79203852615307757112,
                   0.79959221481843290036, 0.90154689901948148467 ),
                nrow=2, ncol=2, byrow=TRUE )
> lhs %*% rhs
                         [,1]                  [,2]
[1,] 1.2021775173654667412 1.1954267253835633156
[2,] 1.5655351790492570974 1.4857111297415830364
```

Листинг 2 — Пример вычисления на языке R

Такое расхождение неслучайно. Если еще раз обратить внимание на листинг 1, то можно увидеть, что в численном эксперименте участвуют матрицы, все элементы которых имеют тип **double**. Согласно спецификации языка C++ [11], для представления чисел с плавающей запятой существуют несколько типов данных, характеристики которых приведены в таблице 3.

Согласно говорящим именам стандартных типов данных, а также информации из приведенной таблицы, тип **double** имеет точность, в два раза превышающую **float**. В общем случае **double** имеет 15–16 десятичных знаков точности, в то время как **float** только 7.

Таблица 3 — Размер и диапазон чисел с плавающей запятой в языке C++

Тип	**Размер, байт**	**Допустимый диапазон значений**
float	4	+/– 3.4e +/– 38 (точность ∼7 цифр)
double	8	+/– 1.7e +/– 308 (точность ∼15 цифр)
long double	8	+/– 1.7e +/– 308 (точность ∼15 цифр)

В численном эксперименте использовался тип данных **double**, что вызвало ошибку округления примерно на 15-м знаке после запятой. Соответственно, если применять тип данных с меньшей (**float**) или большей (**long double**) точностью, меняться будет только положение десятичного знака, на котором будет начинаться расхождение, но *ошибка при этом не исчезнет*.

В такой ситуации разумным видится использование специализированных библиотек, таких как GMP, поддерживающих более точные вычисления. Именно свободный доступ, минимальные ограничения на использование, а также основная функция – поддержка арифметики произвольной точности – делают библиотеку GMP идеальным инструментом для использования в данной работе, особенность которой состоит в оперировании матрицами больших размерностей, что налагает дополнительные (более строгие) ограничения на точность полученного результата.

2.2 Распараллеливание матричных операций

2.3 Тестирование

Тестирование – важная часть процесса разработки программных продуктов, поскольку ошибки – это не случайное явление. Они – обязательный спутник любого проекта.

Различают две формы тестирования. *Тестирование методом прозрачного ящика* (white box testing), в котором испытатель имеет понятие о внутренней работе программы, и *тестирование с алгоритмом типа черного ящика* (black box testing), когда программа тестируется без использования информации о ее реализации. Обе формы тестирования имеют важное значение для проектов, претендующих на попадание в категорию высококачественных. Чаще всего используется тестирование по принципу черного ящика, поскольку оно, как правило, моделирует типичное поведение пользователя. Например, при «черном» тестировании компоненты интерфейса можно рассматривать как кнопки. Если испытатель, щелкнув по кнопке, не увидел никакой реакции на свое действие, он делает вывод о том, что, очевидно, в программе есть ошибка.

Тестирование по принципу черного ящика не может охватить все аспекты испытываемого продукта. Современные программы слишком велики, чтобы реализовать имитацию щелчков по каждой кнопке, проверить все возможные варианты входных данных и выполнить все комбинации команд. Необходимость «белого» тестирования объясняется тем, что гораздо проще гарантировать тестовое покрытие множества неисправностей, если тесты будут написаны на уровне объекта или подсистемы. К тому же, зачастую «белые» тесты легче написать и автоматизировать, чем «черные». В этой работе делается акцент на методах тестирования путем применения «прозрачного ящика», поскольку программист может использовать эти методы еще во время разработки своей программы.

Единственный способ выявить ошибки в программе – провести ее тестирование. Одним из самых важных видов тестирования с точки зрения разработчика является блочное, или поэлементное [20]. Поэлементные тесты представляют собой программы, которые проверяют работоспособность клас-

са или подсистемы. В идеальном варианте для каждой задачи низкого уровня должен существовать один или несколько тестов этого ранга.

Хорошо написанные тесты служат защитой во многих отношениях. Во-первых, они доказывают, что данная часть программы действительно работает должным образом. До тех пор, пока не будет получен код, который и в самом деле оправдывает существование тестируемого класса, его поведение можно считать неизвестным. Во-вторых, поэлементные тесты первыми подают сигнал тревоги, если после недавнего изменения что-то «сломалось». В-третьих, используемые как часть общего процесса разработки, они заставляют разработчика устранять проблемы с самого начала. Если какой-либо участок кода вообще не контролируется с помощью блочных тестов, то в случае возникновения проблемы ее источник можно смело искать в этом участке. В-четвертых, поэлементные тесты позволяют испытать код до объединения с другим кодом. И наконец, поэлементные тесты показывают пример применения созданного кода.

Чем больше будет написано тестов, тем более обширным тестовым покрытием будет обладать код. Чем шире тестовое покрытие, тем меньше вероятность, что ошибки останутся невыявленными.

Существуют разные методики написания поэлементных тестов. Методология экстремального программирования (Extreme Programming methodology) предписывает своим сторонникам создавать тесты еще до написания кода. Теоретически предварительное написание тестов помогает четче сформулировать требования к компоненту и предложить систему показателей, которые могут быть использованы для определения момента завершения кода. Менее жесткий вариант состоит в проектировании тестов до кодирования, но в расчете на более позднюю их реализацию. В этом случае программист попрежнему вынужден четко понимать требования, предъявляемые к модулю, но не обязан писать код, использующий еще несуществующие классы.

Код теста во многом зависит от типа используемой тестовой оболочки. В данной работе используется оболочка **CppUnit** – это программный набор инструментов для модульного тестирования программ на языках программи-

рования C/C++ [10]. Система разрабатывается с 2000 г. и публикуется под GNU LGPL.

Оболочка запускает *тесты*, сгруппированные в *пакеты*. Результаты тестирования направляются в указанные фильтры, набор которых разнообразен: от простейшего подсчета пройденных либо не пройденных тестов до более продвинутых, поддерживающих XML-вывод, совместимый с дальнейшей интеграцией с системами отчетов.

Для того, чтобы понять, каким образом работает система тестирования `CppUnit`, рассмотрим, как спроектирован класс `MatrixTest`, основная задача которого – проведение блочного тестирования класса `Matrix`.

Обратим внимание на то, как объявлен класс в заголовочном файле (полный код класса приведен в приложении !TODO: REF TO APPENDIX).

```
22 class MatrixTest : public CppUnit::TestFixture
23 {
```

Видно, что класс `MatrixTest` наследуется от класса `TestFixture`. Класс `TestFixture` – это главная «игровая фигура» оболочки `CppUnit`; он представляет собой логическую группу тестов.

Оболочка `CppUnit` выполняет группу тестов в виде некоторого набора (тестового комплекта), который должен содержать информацию о том, какие тесты подлежат выполнению, а какие нет (в отличие от класса `TestFixture`, который просто логически группирует тесты). Для этой цели `CppUnit` предоставляет класс `TestSuite`, который содержит в себе произвольный набор тестов. Обычная практика использования класса `TestSuite` состоит в следующем. В классы тестов (например такие, как `MatrixTest`), добавляется специальный статический метод.

```
1 public:
2   static CppUnit::Test* suite()
3   {
4     ...
5   }
```

После чего, посредством вызова этого метода, происходит добавление произвольного количества тестов в тестовый набор и запуск этого набора на выполнение. Как можно заметить, это довольно рутинная, повторяющаяся для каждого нового теста, задача. Поэтому `CppUnit` предоставляет набор

вспомогательных макросов, которые призваны упростить процесс создания наборов тестов.

Поскольку в данной работе разрабатывается класс, представляющий собой описание математического объекта (матрицы) в программе, для проверки его работоспособности видится естественным проведение как минимум следующих тестов:

- Тест сложения матриц;
- Тест умножения матриц;
- Тест транспонирования матриц.

Двигаясь далее по исходному коду класса `MatrixTest`, находим строки, где происходит объявление перечисленных тестов.

```
24  CPPUNIT_TEST_SUITE( MatrixTest );
25    CPPUNIT_TEST( testAddition );
26    CPPUNIT_TEST( testMultiplication );
27    CPPUNIT_TEST( testTransposition );
28    CPPUNIT_TEST( testTime );
29  CPPUNIT_TEST_SUITE_END();
```

Это один из примеров использования вспомогательных макросов. Такой код осуществляет автоматическое создание набора тестов `MatrixTest` и добавление тестов `testAddition`, `testMultiplication`, `testTransposition` и `testTime` в набор для последующего выполнения.

Далее, происходит объявление соответствующих методов класса.

```
31  public:
32
33    void testCrashDoubleType();
34    void testAddition();
35    void testMultiplication();
36    void testTransposition();
37    void testTime();
```

Сразу возникает вопрос, почему в списке появился тест `testCrashDoubleType`, который не был добавлен в набор выше? Ответ прост и кроется уже в названии теста. Этот тест демонстрирует появление ошибки округления, возникающей при использовании типа `double` и проводит к систематическому непрохождению тестового набора. По этой причине он был исклю-

чен. Исходный код этого теста уже был подробно проанализирован в этой работе (см. листинг 1 и комментарии к нему).

В оставшейся закрытой части класса осуществляются объявления вспомогательных данных, участвующих в проведении тестов.

```
39 private:
40
41   typedef mpz_class element_t;
42   typedef Matrix< element_t > matrix_t;
43
44   std::size_t matrixSize = 50;
45
46   MatrixRandomFiller matrixRandomFiller;
47   MatrixSummarizer matrixSummarizer;
48   MatrixMultiplier matrixMultiplier;
49   MatrixTransposer matrixTransposer;
50
51   TimeMeasurer timeMeasurer;
52 };
```

В строке 41 создается новый тип данных `element_t`, который является простым синонимом к типу `mpz_class`, определенному в библиотеке GMP и отвечающему за арифметику расширенной точности. В строке 42 определяется тип `matrix_t`, представляющий собой матрицу, элементы которой имеют тип `element_t`. В строке 44 задается размерность матриц, участвующих в тестах. В строках 46–51 происходит создание объектов вспомогательных классов. Благодаря их говорящим названиям, нетрудно догадаться, для чего они были созданы: `MatrixRandomFiller` заполняет матрицу случайными значениями, `MatrixSummarizer` складывает, `MatrixMultiplier` умножает, а `MatrixTransposer` транспонирует матрицы. Класс `TimeMeasurer` отвечает за замер времени выполнения участка кода.

Чтобы выполнить набор тестов и увидеть результаты, необходим *прогонщик тестов*. В оболочке `CppUnit` содержится несколько различных прогонщиков, которые действуют в различных средах (например, прогонщик `MFCRunner` предназначен для выполнения в программе, написанной с использованием библиотеки базовых классов Microsoft). Для текстовых сред используется прогонщик `TextRunner`. Ниже приводится код тела функции

`main`, в котором происходит создание всех основных объектов тестового окружения.

```
14 int main( int argc, const char* argv[ ] )
15 {
16   CppUnit::TestResult controller;
17
18   CppUnit::TestResultCollector result;
19   controller.addListener( &result );
20
21   CppUnit::TextTestProgressListener progress;
22   controller.addListener( &progress );
23
24   CppUnit::TextUi::TestRunner runner;
25   runner.addTest( CppUnit::TestFactoryRegistry::getRegistry().makeTest() );
26
27   runner.run( controller, "" );
28
29   CppUnit::CompilerOutputter outputter( &result, std::cerr );
30   outputter.write();
31
32   return result.wasSuccessful() ? 0 : 1;
33 }
```

В строке 16 создается объект `controller` класса `TestResult`, представляющий собой статистику прохождения тестов. Другие вспомогательные классы будут подключаться именно к нему. В строке 18 создается сборщик статистики. В строке 21 – объект, следящий за ходом выполнения всего набора тестов. В строке 24 создается прогонщик тестов и запускается в строке 27. В строке 29 создается объект, отвечающий за печать результатов теста в совместимом с компилятором формате.

Рассмотрим структуру самих тестов, описанных в файле реализации. Для регистрации созданного набора тестов в системе используется такая строка.

```
14 CPPUNIT_TEST_SUITE_REGISTRATION( MatrixTest );
```

Для проверки утверждений и определения точного места, где они не выполняются, используется макрос `CPPUNIT_ASSERT`. Рассмотрим первый тест, где он применяется.

```
49 void MatrixTest::testAddition()
50 {
51   matrix_t A( this->matrixSize, this->matrixSize );
```

```
52    this->matrixRandomFiller.fill( A );
53
54    matrix_t B( this->matrixSize, this->matrixSize );
55    this->matrixRandomFiller.fill( B );
56
57    matrix_t C( this->matrixSize, this->matrixSize );
58    this->matrixSummarizer.summarize( C, A, B );
59
60    matrix_t D( B + A );
61
62    CPPUNIT_ASSERT( C == D );
63  }
```

Это *тест, проверяющий корректность сложения матриц*. В строках 51–55 с помощью уже известных объектов вспомогательных классов происходит создание и заполнение операндов, участвующих в тесте. В строке 58 осуществляется вычисление результата сложения двух операндов сторонним классоми и сохранение этого результата в переменной `C`. В строке 60 сложение выполняет сам класс `Matrix`, сохраняя результат в переменную `D`. В строке 62 осуществляется формальная проверка на равенство двух независимо полученных результатов. В этой точке теста мы *ожидаем*, что значения этих результатов совпадут. Если окажется, что это не так, `CppUnit` сообщит о провале прохождения теста и укажет на эту точку, сообщив имя файла и номер строки.

Тест матричного умножения во многом схож с первым тестом. Его исходный код приведен ниже.

```
67  void MatrixTest::testMultiplication()
68  {
69    matrix_t A( this->matrixSize, this->matrixSize );
70    this->matrixRandomFiller.fill( A );
71
72    matrix_t B( this->matrixSize, this->matrixSize );
73    this->matrixRandomFiller.fill( B );
74
75    matrix_t C( this->matrixSize, this->matrixSize );
76    this->matrixMultiplier.multiply( C, A, B );
77
78    matrix_t D( A * B );
79
80    CPPUNIT_ASSERT( C == D );
```

```
81 ||
82 ||   matrix_t E( A );
83 ||   element_t value = 2;
84 ||   this->matrixMultiplier.multiply( value, E );
85 ||
86 ||   matrix_t F( value * A );
87 ||
88 ||   CPPUNIT_ASSERT( A == E );
89 || }
```

Единственное отличие состоит в дополнительной проверке в строках 82–88 работы операции умножения матрицы на число.

Тест транспонирования необхоодим для контроля правильности выполнения соответствующей матричной операции.

```
93 || void MatrixTest::testTransposition()
94 || {
95 ||   matrix_t matrix( this->matrixSize, 2 * this->matrixSize );
96 ||   this->matrixRandomFiller.fill( matrix );
97 ||
98 ||   matrix_t matrix_copy = matrix;
99 ||
100||   matrix_t matrix_transp( matrix.get_columns(), matrix.get_rows() );
101||   this->matrixTransposer.transpose( matrix_transp, matrix );
102||
103||   matrix.transpose();
104||
105||   CPPUNIT_ASSERT( matrix == matrix_transp );
106||
107||   const std::size_t row = 4;
108||   const std::size_t column = 5;
109||
110||   CPPUNIT_ASSERT( matrix_copy( row, column ) == matrix( column, row ) );
111||
112||   matrix.transpose();
113||
114||   CPPUNIT_ASSERT( matrix == matrix_copy );
115||   CPPUNIT_ASSERT( matrix( row, column ) == matrix_copy( row, column ) );
116|| }
```

Рассмотрим тест подробнее. В строке 95 создается случайно заполненная матрица, и ее первоначальное значение в строке 98 дублируется. В строке 101 сторонний класс транспонирует исходную матрицу, сохраняя результат в переменной `matrix_transp`. В строке 103 исходную матрицу транспони-

рует сам класс, после чего, в строке 105, результат операции сразу проверяется. В строке 110 производится проверка доступа по индексу. Смысл этой операции в том, чтобы убедиться в свойстве зеркальности операции транспонирования, а именно что (i, j)-й элемент исходной матрицы соответствует (j, i)-му элементу транспонированной. В строке 112 матрица транспонируется обратно, и в строке 114 проверяется на соответствие своему первоначальному значению. В строке 115 осуществляется еще одна проверка индексного доступа.

Таким образом, были рассмотрены все блочные тесты, участвующие в проверке корректности работы разрабатываемого класса `Matrix`. Их регулярное прохождение будет доказывать правильность работы каждого участка кода. А в случае возникновения ошибки оболочка `CppUnit` позволит быстро локализовать и устранить проблему.

Выводы по главе два

Все выводы по этой главе съел безумный барсук.

Заключение

В данной работе были приведены основные положения, выведенные из идей физической экономики. На основе этих положений была построена модель экономической системы древнего общества скотоводов-земледельцев.

В дипломной работе выполнены следующие **задачи**:

1) построена имитационная модель экономической системы древнего общества скотоводов-земледельцев;
2) проведен имитационный эксперимент;
3) проверены теоретические положения.

Можно сделать следующие выводы после проведения симуляционного эксперимента в среде VisSim:

1) модель экономической системы древнего общества скотоводов-земледельцев является жизнеспособной и правдоподобно описывает поведение древней человеческой общины.
2) модель дает возможность объяснить экономическую сущность исторических фактов относительно древних общин периода неолита;
3) проверен принцип увеличения доли свободного времени в общем фонде социального времени по ходу развития общины.

Данная модель может быть улучшена путем более точного описания различных хозяйственных процессов, происходивших в экономике общины. Также возможно применение тензорной методологии для описания этих хозяйственных процессов, при этом уравнения примут более понятный внешний вид, не утратив своего содержания.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Boehm, B. W. Industrial software metrics top 10 list / Barry W. Boehm // IEEE Software 4. — 1987. — no. 9 (September). — 84–85.
2. Fork–join model . — URL: http://en.wikipedia.org/wiki/Fork-join_model (дата обращения: 12.04.2015).
3. The gnu multiple precision arithmetic library . — URL: https://gmplib.org/ (дата обращения: 15.04.2015).
4. Held, M. The traveling salesman problem and minimum spanning trees. part. 2. / M. Held, R. Karp // Math. Program. — 1971. — 1, N 1, p. 6–25.
5. Held, M. Validation of subgradient optimization. / M. Held, P. Wolfe, H. Crowder // Math. Program. — 1974. — 6, N 1, p. 62–88.
6. Huang, H. Unified approach to quadratically convergent algorithms for function minimization / H. Huang. — J. Optimizat. Theory and Appl., 1970. — 5, N 6.
7. Knuth, D. An empirical study of fortran programs / Donald Knuth // Software – Practice and Experience. — 1971. — 105–33.
8. Mathematical programming. Study 3. Nondifferentiable optimization / Ed. by M. L. Balinski, P. Wolfe. — Amsterdam: North–Holl and Publ. co., 1975. — 178 p.
9. McCool, M. Structured Parallel Programming: Patterns for Efficient Computation / Michael McCool, James Reinders, Arch Robinson. — MK, 2013.
10. Mohrhard, M. CppUnit Documentation / Markus Mohrhard. — freedesktop.org.
11. Stroustrup, B. The C++ Programming Language / Bjarne Stroustrup. — Addison-Wesley, 2013.
12. Wolframalpha . — URL: http://www.wolframalpha.com (дата обращения: 10.04.2015).
13. Вирт, Н. Алгоритмы и структуры данных. Новая версия для Оберона + CD / Пер. с англ. Ткачев Ф. В. / Н. Вирт. — М.: ДМК Пресс, 2010. — 272 с.: ил.

14. Гершович, В. И. Метод эллипсоидов, его обобщения и приложения / В. И. Гершович, Н. З. Шор // Кибернетика. — 1982. — №5.
15. Грин, Д. Математические методы анализа алгоритмов / Д. Грин, Д. Кнут. — М.: Мир, 1987. — 120 с., ил.
16. Данилин, А. И. Основы теории оптимизации (постановки задач) [Электронный ресурс] : электрон. учеб. пособие / А. И. Данилин. — Минобрнауки России, Самар. гос. аэрокосм. ун-т им. С.П. Королева (нац. исслед. ун-т). – Электрон. текстовые и граф. дан. (1,2 МБайт). – Самара, 2011. — 1 эл. опт. диск (CD-ROM).
17. Ермольев, Ю. М. Методы стохастического программирования / Ю. М. Ермольев. — М.: Наука, 1976. — 240 с.
18. Макконнелл, С. Совершенный код. Мастер-класс / Пер. с англ. / С. Макконнелл. — М.: Издательство «Русская редакция», 2010. — 896 стр.: ил.
19. Поляк, Б. Т. Один общий метод решения экстремальных задач. – Докл. АН СССР / Б. Т. Поляк. — 1967. — 174, № 1, с. 33–36.
20. Солтер, Н. А. C++ для профессионалов.: Пер. с англ. / Николас А. Солтер, Скотт Дж. Клеппер. — М.: ООО «И.Д. Вильямс», 2006. — 912 с.: ил. – Парал. тит. англ.
21. Стецюк, П. И. Методы эллипсоидов и r-алгоритмы / П. И. Стецюк. — Нац. акад. наук Украины, Ин-т кибернетики им. В. М. Глушкова, Акад. транспорта, информатики и коммуникаций. – Кишинэу: Эврика, 2014. — 488 с.
22. Хачиян, Л. Г. Полиномиальные алгоритмы в линейном программировании / Л. Г. Хачиян // Ж. вычисл. матем. и матем. физ. — 1980. — С. 51–68.
23. Шор, Н. З. Применение метода градиентного спуска для решения сетевой транспортной задачи. – В кн.: Материалы науч. семинара по теорет. и прикл. вопр. кибернетики и исслед. операций / Н. З. Шор. — Науч. совет по кибернетике АН УССР. Киев, 1962. — вып. 1, с. 9–17.

24. Шор, Н. З. О структуре алгоритмов численного решения задач оптимального планирования и проектирования: Автореф. дис. ... канд. физ.-мат. наук. / Н. З. Шор. — Киев, 1964. — 10 с.
25. Шор, Н. З. Методы минимизации недифференцируемых функций и их приложения: Автореф. дис. ... докт. физ.-мат. наук. / Н. З. Шор. — Киев, 1970. — 44 с.
26. Шор, Н. З. О методе минимизации почти дифференцируемых функций / Н. З. Шор // Кибернетика. — 1972. — № 4, с. 65–70.
27. Шор, Н. З. Метод отсечения с растяжением пространства для решения задач выпуклого программирования / Н. З. Шор // Кибернетика. — 1977. — № 1, с. 94–95.
28. Шор, Н. З. Методы минимизации недифференцируемых функций и их приложения / Н. З. Шор. — Киев: Наук. думка, 1979. — 200 с.
29. Юдин, Д. Б. Информационная сложность и эффективные методы решения выпуклых экстремальных задач / Д. Б. Юдин, А. С. Немировский // Экономика и мат. методы. — 1976. — т. 12, вып. 2. – С. 357–369.